**Journal of Applied Informatics and Computing (JAIC)**
Vol.3, No.1, Juli 2019, pp. 01~07
e-ISSN: 2548-6861 

# Desain Mesin *Compiler* untuk Penganalisa Leksikal, Sintaksis, Semantik, Kode Antara dan *Error Handling* Pada Bahasa Pemrograman Sederhana

**Andez Apriansyah [1]*, Tri Ichsan Saputra [2]*, Fauziah [3]*, Yunan Fauzi Wijaya[4]***
* Informatika, Universitas Nasional
andezapriansyah@gmail.com [1], triichsan1@gmail.com [2], fauziah@civitas.unas.ac.id [3], yunanfw@gmail.com

**Article Info**

***Article history:***

Received 04-03-2019
Revised 25-03-2019
Accepted 01-04-2019

***Keyword:***

*Error Handling,*
*Intermediate Code,*
*Lexical,*
*Semantic,*
*Syntactic.*

**ABSTRACT**

In compilation techniques, the processes and stages carried out to relate to translating source languages into target languages (object programs). Source languages are high-level programming languages that are easy to understand and easy to learn by humans, while target languages are low-level languages that are only understood by machines. In this study, a compiler machine called Automatic LESSIMIC Analyzer is used which can be used to analyze, including lexical, syntactic, and semantic analysis. Compiler machines that are designed can also synthesize intermediate code, using assembler codes. The compiler engine will produce an analysis of the program code that the user enters in the form of an error message if the program code is not in accordance with the grammar that applies generally in programming languages. In this research, the simple program code that is inputted is C ++ programming language, and successfully analyzes the lexical, semantic, syntactic, intermediate code generation and successfully detects errors from the source program.

## I. PENDAHULUAN

Dalam proses atau tahapan yang dilakukan pada teknik kompilasi, diantaranya adalah mempelajari sebuah proses untuk menerjemahkan bahasa sumber menjadi bahasa target. Alat atau *software* yang digunakan untuk proses kompilasi disebut juga dengan *compiler* (kompilator). *Compiler* adalah alat bantu untuk melakukan proses terhadap suatu bahasa sumber menjadi bahasa target. Bahasa sumber yang digunakan adalah bahasa pemrograman tingkat tinggi, dan bahasa target atau bahasa sasaran berupa kode antara yang kemudian diterjemahkan menjadi bahasa mesin [1].

Selain melakukan proses transformasi terhadap suatu bahasa sumber menjadi bahasa target, *compiler* juga dapat menemukan pesan *error* yang bisa memudahkan *programmer*



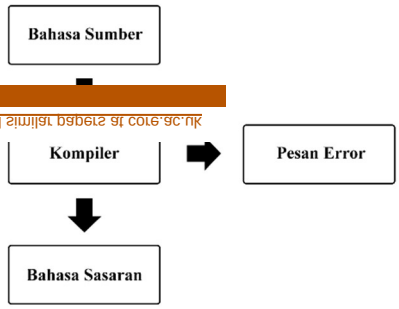
*Compiler* ini juga bisa sebagai alat bantu proses pembelajaran dalam menentukan kode antara dan mengkoreksi bahasa pemrograman.

Oleh karena itu, dalam penelitian ini kami mendesain sebuah *compiler* sederhana dengan program sumber pada bahasa pemrograman C++, yang didalamnya terdapat analisis leksikal, semantik, sintaksis, tabel simbol, pesan *error*, dan kode antara.

## II. LANDASAN TEORI

### A. *Compiler*

*Compiler* adalah perangkat lunak aplikasi sistem yang melakukan proses transformasi terhadap suatu program sumber menjadi program target. Dalam hal ini, program sumber yang digunakan adalah bahasa pemrograman tingkat tinggi, dan program targetnya berupa bahasa *assembly*. Sebagai penterjemah, *compiler* harus mampu membimbing penggunanya apabila terdapat kesalahan dalam program sumber [1].




Gambar 1. Konsep Dasar Teknik Kompilasi

*B. Grammar*

*Grammar* ialah aturan-aturan pembentukan suatu kalimat dalam sebuah bahasa, atau yang biasa disebut tata bahasa. Dengan adanya *grammar*, *parsing* dapat dilakukan secara cepat. *Parser* akan mencari aturan-aturan yang tepat untuk membentuk struktur suatu kalimat (dalam penelitian ini bahasa pemrograman) [2].

*C. Analisis dan Sintesis*

Proses *compiler* dapat dibagi menjadi dua bagian utama yaitu [3]:

*1) Analisis*: Pada tahap analisis, program yang ditulis dalam bahasa sumber dibagi dan dipecah ke dalam beberapa bagian yang kemudian akan direpresentasikan ke dalam suatu bentuk dari program sumber. Pada tahap ini operasi-operasi yang dilakukan oleh program sumber ditentukan dan dicatat dalam suatu struktur pohon (*tree*) yang disebut dengan nama pohon [3].

*2) Sintesis*: Program sasaran dibentuk berdasarkan representasi antara yang dihasilkan pada tahap analisis. Selain itu compiler perlu juga editor penghubung melakukan dua hal yang berhubungan dengan *loading* dan *link-editing*. Proses pembacaan (*loading*) dilakukan pengambilan kode mesin yang kemudian perintah dan data yang diperoleh diletakkan pada memori tempat yang seharusnya [3].

*D. Penanganan Kesalahan*

Penanganan kesalahan (*error handling*) dilakukan bila terjadi kesalahan dalam penulisan program sumber, baik kesalahan penulisan besaran leksikal, kesalahan sintaksis, maupun kesalahan semantik [4].

## III. METODE PERANCANGAN APLIKASI

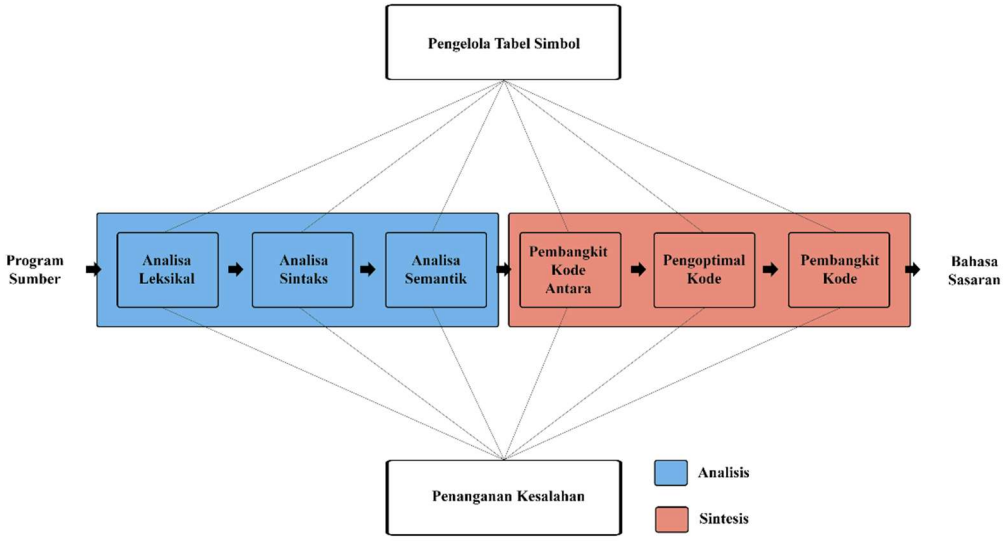
Tahapan yang dilakukan dalam perancangan aplikasi pada proses desain *compiler* untuk bahasa pemrograman sederhana. Metode perancangan aplikasi yang digunakan dalam penelitian ini berkaitan dengan mesin *compiler* yang didesain dengan memperhatikan fase atau tahapan yang ada pada proses kompilasi, dapat dilihat pada Gambar 2 mengenai alur kompilasi yang terjadi didalam *compiler*.

Gambar 2 menjelaskan alur sistem perancangan aplikasi dengan tahapan sebagai berikut:

1. *User* menginputkan kode program dengan bahasa tingkat tinggi, misalnya dengan bahasa pemrograman C++
2. Setelah kode program diinputkan ke dalam aplikasi compiler yang telah di desain, maka mesin dapat melakukan hasil *output* berupa:
   a. Analisis leksikal
   b. Analisis semantik
   c. Analisis sintaksis
   d. Selain melakukan tahapan analisis leksikal, semantik, sintaksis, mesin dapat menghasilkan tabel simbol dan kode antara serta *message error*.

Metode yang digunakan dalam pengaplikasian proses/alur kompilasi yaitu metode SDLC (*System Development Life Cycle*).


Gambar 3. Metode SDLC Untuk Proses Kompilasi




Gambar 2. Alur Kompilasi Pada Mesin *Compiler*

Metode SDLC pada Gambar 3 dapat dijelaskan, sebagai berikut:

1. Analisis Kebutuhan
   Kebutuhan baik itu *hardware* maupun *software* yang diperlukan dalam membuat aplikasi diantaranya :
   a. Hardware :
      - Laptop dengan minimal RAM 4GB, ROM space 100MB dan processor quad-core 2.2 GHz.
      - Mouse general electric.
   *b.* Software : Visual Studio 2010 untuk proses pembuatan interface *Automatic LESSIMIC Analyzer*
2. Desain
   Desain tampilan menggunakan beberapa *tools* yang terdiri dari Button, TextBox, Label, Group Box. Kode program pada aplikasi *Automatic LESSIMIC Analyzer* yang digunakan pada penelitian ini adalah Bahasa pemrograman C#.
3. Implementasi
   Proses implementasi dilakukan dengan menggunakan pengujian *Black Box.*
4. *Testing*
   Dalam melakukan *testing* pada aplikasi ini menggunakan bahasa pemrograman C++.
5. Evaluasi
   Tahapan yang dilakukan setelah proses testing dan implementasi berhasil dan memperbaiki segala kesalahan yang ada pada aplikasi yang didisain.

## IV. HASIL DAN PEMBAHASAN

### *A. Analisis*

Analisis dalam penelitian ini terbagi menjadi 3 (tiga) bagian yang mana ketiga analisis berikut memiliki tugasnya masing-masing, diantaranya:

*1) Analisis Leksikal (Scanner)* melakukan pemeriksaan terhadap kode sumber dengan perubahan status yang terjadi. Dalam penerapannya, suatu kode sumber akan diuraikan menjadi simbol-simbol tertentu yang disebut token. *Scanner* berperan sebagai antarmuka antara *source code* dengan proses analisis sintaksis (*parser*). *Scanner* akan memeriksa setiap karakter dari kode sumber [4].

Analisis leksikal berkaitan dengan penulisan bahasa pemrograman (dalam penelitian yaitu bahasa C++), misalnya, cout, cin, float, dll.

*2) Analisis Sintaksis (Parser)*: Bahasa pemrograman dibentuk oleh aturan yang dapat direpresentasikan dengan struktur sintaks. Dari penggunaan sintaks sebuah bahasa pemrograman dapat dibangun pohon urai (*parse tree*) seiring dengan proses *parser*. *Parser* menerima masukan dari *scanner* (dalam bentuk *token*) dan membentuk *parse tree* sesuai dengan sintaks dan tata bahasanya. Jika sekumpulan *token* yang diterima *parser* tidak sesuai dengan aturan tata bahasa pembentuk bahasa pemrograman, maka terjadi kesalahan sintaks [4]. Penelitian lain yang telah dilakukan sebelumnya yaitu, penggunaan informasi sintaksis di dalam teks dan mengenali kesamaan kata dengan WordNet dan database leksikal [5].

Analisis sintaksis berkaitan dengan logika matematika, misalnya, a=b+c, x=1*a, dll.

*3) Analisis Semantik* memanfaatkan pohon sintaks yang dihasilkan pada proses parsing. Secara umum, fungsi dari analisis semantik adalah menentukan makna dari serangkaian instruksi yang terdapat dalam program sumber [6].

Analisis semantik berkaitan dengan variabel dalam kode program, misalnya variabel total, jumlah, selisih, dll.

### *B. Tabel Simbol*

Pada mesin *Automatic LESSIMIC Analyzer* terdapat 5 (lima) buah bagian dari tabel simbol, yang mana dapat dilihat pada Tabel 1. Tabel simbol berfungsi untuk mendeklarasikan semua input berdasarkan kode program yang di *compile* [7].

TABEL 1
Tabel Simbol dalam Kompilasi

| Tabel Simbol | Contoh |
|---|---|
| Identifier | a, t, c |
| Keyword | include, iostream, cout |
| Tipe Data | int, float, char |
| Operator | (=) (+) (-) |
| Delimeter | ( ; { |

### *C. Notasi N-Tuple pada Kode Antara*

Notasi *N-Tuple* terdiri dari:

*1) Triples Notation*

Format pada *triples notation* yaitu:
<operator> <operand> <operand>
Contoh intruksi: A := X + Y / Z
Maka, didapat kode antara triple sebagai berikut:

1. / , Y , Z
2. + , X , (1)
3. := , A , (2)

Untuk operator / (pembagian) ataupun jika nantinya ditemukan operator * (perkalian) akan diprioritaskan [8].

Kekurangan dari notasi *triple* sulit saat melakukan optimasi pemecahannya dengan *Indirect Triples* yang memiliki dua *list* yaitu *list* intruksi dan *list* eksekusi [8].

Misal:
X := A + B / C
Y := B / C

List Intruksi:
1. / , B , C
2. + , (1) , A
3. := , X , (2)

4. := , Y, (1)

List Eksekusi:
1. 1
2. 2
3. 3
4. 1
5. 4

*2) Quadruples Notation*

Format pada *quadruples notation* yaitu:
<operator> <operan> <operan> <hasil>
Contoh intruksi:
X := (A + B) / (C - D)
Maka bentuk *quadruples* sebagai berikut:
1. + , A , B , H1
2. - , C , D , H2
3. / , H1 , H2 , X

## D. Pembangkit Kode

Pembangkit kode atau *code generator* difungsikan untuk mentranslasikan dari kode antara ke bahasa *assembly* atau bahasa mesin [8].
Contoh:
(A - B) * C / D
Bentuk *quadruples* :
1. , A , B , H1
2. * , H1 , C , H2
3. / , H2 , D , H3

Translasi bahasa *assembly* (sebelum optimasi) :

```
LDA A
SUB B
STO H1
LDA H1
MUL C
STO H2
LDA H2
DIV D
STO H3
```

## E. Optimasi Kode

Kemudian hasil pembentukan kode diatas dapat dilakukan optimasi supaya lebih efisien, sehingga pengulangan suatu ekspresi tidak perlu terjadi [8].

Kode *assembly* pada translasi diatas dapat dioptimasi menjadi :

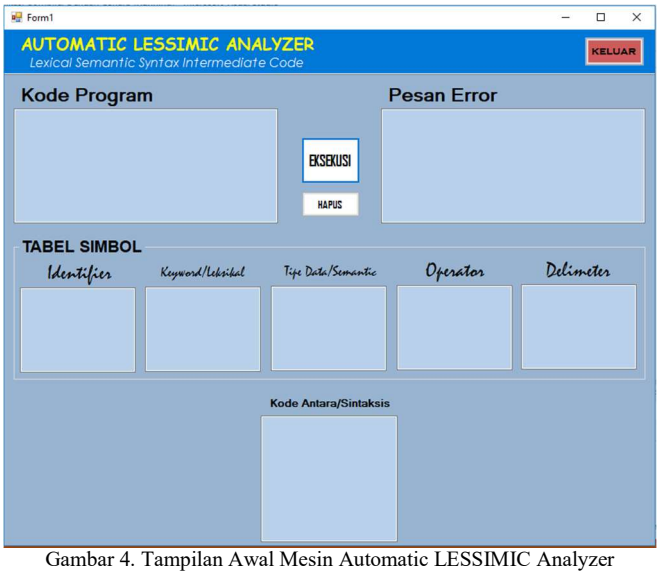
```
LDA A
SUB B
STO H1
LDA C
DIV D
MUL H1
STO H2
```

## F. Error Handling

Pada aplikasi *Automatic LESSIMIC Analyzer*, *compiler* akan menangani kesalahan yang terjadi pada *source code*, bentuk penanganan kesalahan tersebut berupa *message error*. *Message error* akan menampilkan bentuk kesalahan dari proses analisa, diantaranya analisa leksikal, sintaksis dan semantik.

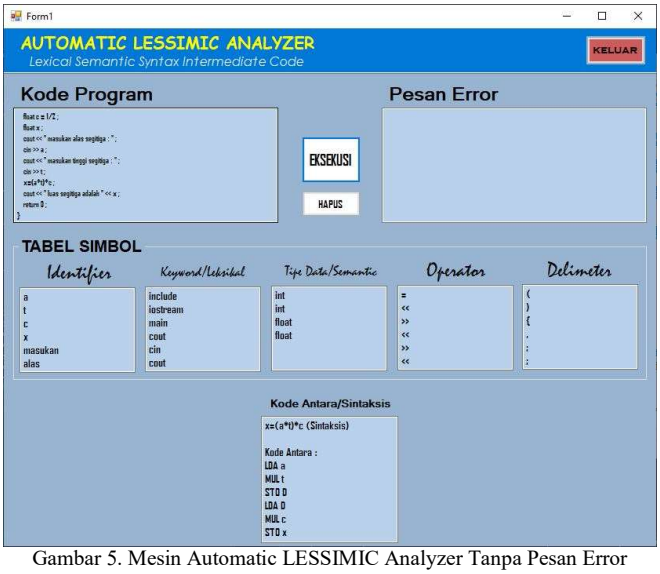
## G. Tampilan Mesin Automatic LESSIMIC Analyzer

Rancangan *user interface* dari *Automatic LESSIMIC Analyzer* bisa dilihat dari Gambar 4. Terdapat 8 kolom yang terdiri dari kode program, pesan *error*, kode antara, tabel identifier, tabel *keyword*, tabel tipe data, tabel operator, tabel delimeter.



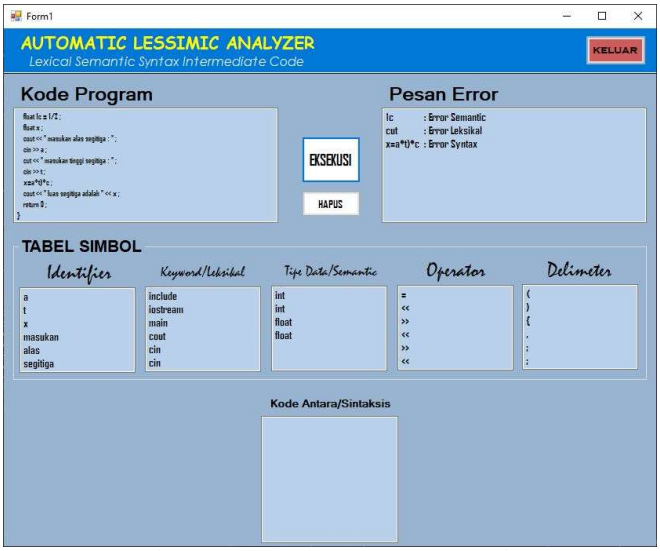
Gambar 4. Tampilan Awal Mesin Automatic LESSIMIC Analyzer

Jika *user* menginputkan kode program dan melakukan eksekusi, kemudian *compiler* akan menganalisa dan jika kode program tersebut tidak mengalami *error*, maka akan didapatkan hasil seperti Gambar 5.




Gambar 5. Mesin Automatic LESSIMIC Analyzer Tanpa Pesan Error

Pada Gambar 5 dapat dilihat bahwa *textbox* Pesan *Error* tidak memiliki isi apapun (kosong), mengartikan bahwa kode program yang diinputkan oleh *user* tidak memiliki *error*. Namun jika kode program yang diinputkan tidak benar, maka *textbox* pada Pesan *Error* akan terisi menyesuaikan pada kesalahan yang terjadi, sebagai contoh dapat dilihat pada Gambar 6.

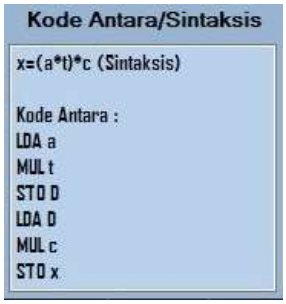
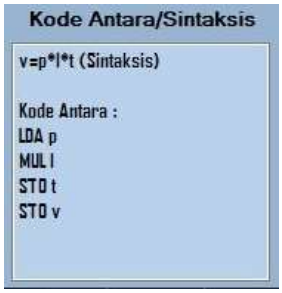
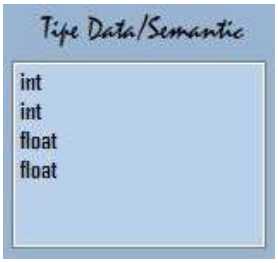
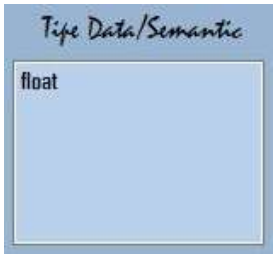


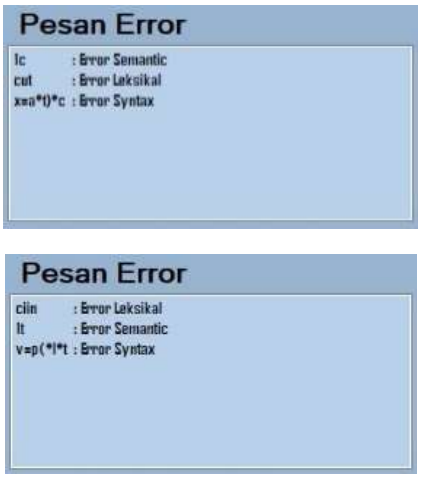

Gambar 6. Mesin Automatic LESSIMIC Analyzer

Hasil yang diperoleh dari penelitian aplikasi dijelaskan pada Tabel 2.

TABEL 2
Hasil Pembahasan Aplikasi Dengan Pengujian *Black Box*

| Fase | Source Code | Aplikasi *Automatic LESSIMIC Analyzer* | Keberhasilan Fungsi Aplikasi |
|---|---|---|---|
| Leksikal (menganalisa bahasa pemrograman) | if (temp2 != "") kumpulankata.Add(temp2);<br>for (int i = 0; i < kumpulankata.Count; i++)<br>{<br>if (_keyword(kumpulankata[i]))<br>{ cekkumpulankata.Add(kumpulankata[i] + "" + System.Environment.NewLine);<br>}<br>}<br>if (error2) { }<br>else<br>{<br>for (int i = 0; i < cekkumpulankata.Count; i++)<br>{<br>result2 += cekkumpulankata[i] + "";<br>}<br>} tabelkey.Text = result2; | Keyword/Leksikal<br>include<br>iostream<br>main<br>cout<br>cin<br>cout<br><br>Keyword/Leksikal<br>include<br>iostream<br>include<br>conio<br>void<br>main | Berhasil |

| | | | |
|---|---|---|---|
| Sintaksis (menganalisa aritmatika) & Kode Antara (mentranslasi bahasa sumber, menjadi bahasa target) | if (temp7 != "") kumpulankata.Add(temp7);<br>for (int i = 0; i < kumpulankata.Count; i++)<br>{<br>if (_kodeantara(kumpulankata[i]))<br>{ cekkumpulankata.Add(kumpulankata[i] + " " + Environment.NewLine");<br>}<br>}<br>if (error7) { }<br>else<br>{<br>for (int i = 0; i < cekkumpulankata.Count; i++)<br>{<br>result7 += cekkumpulankata[i] + "";<br>}<br>} tabel_antara.Text = result7; |  <br>  | Berhasil |
| Semantik (menganalisa variabel) | if (temp6 != "") kumpulankata.Add(temp6);<br>for (int i = 0; i < kumpulankata.Count; i++)<br>{<br>if (_int(kumpulankata[i]))<br>{ cekkumpulankata.Add(kumpulankata[i] + "" + System.Environment.NewLine);<br>}<br>}<br>if (error6) { }<br>else<br>{<br>for (int i = 0; i < cekkumpulankata.Count; i++)<br>{<br>result6 += cekkumpulankata[i] + "";<br>}<br>} tabeltipe.Text = result6; |  <br>  | Berhasil |

| | | | |
|---|---|---|---|
| *Error Handling* (menampilkan pesan terhadap *source code* yang diinputkan, jika terdapat kesalahan dalam analisa leksikal, sintaksis, ataupun semantik) | for (int i = 0; i < kumpulankata.Count; i++)<br>{<br>if (_syntax(kumpulankata[i]))<br>{ cekkumpulankata.Add(kumpulankata[i] + " \t: Error Syntax" + System.Environment.NewLine);<br>}<br>else if (_leksikal(kumpulankata[i]))<br>{<br>cekkumpulankata.Add(kumpulankata[i] + " \t: Error Leksikal" + System.Environment.NewLine);<br>}<br>else if (_semantic(kumpulankata[i]))<br>{<br>cekkumpulankata.Add(kumpulankata[i] + " \t: Error Semantic" + System.Environment.NewLine);<br>}<br>}<br>if (error) { }<br>else<br>{<br>for (int i = 0; i < cekkumpulankata.Count; i++)<br>{<br>result += cekkumpulankata[i] + "";<br>}<br>} tbhasil.Text = result; |  | Berhasil |

Tabel 2 menjelaskan mengenai pembahasan pengujian (uji coba) menggunakan *Automatic LESSIMIC Analyzer* dengan proses pengujian *Black Box*.

Pengujian *Black Box* pada aplikasi *Automatic LESSIMIC Analyzer* merupakan pengujian yang dilakukan untuk mengetahui fungsi yang ada pada aplikasi sudah berjalan sesuai atau tidak.

## V. KESIMPULAN

*Compiler* bertujuan untuk menjalankan proses kompilasi secara keseluruhan, dari hasil pembahasan yang telah di uji coba menggunakan analisa leksikal, sintaksis, semantik juga kode antara dan *error handling*, telah berhasil dengan mengetahui setiap *source code* yang diinputkan menghasilkan *output* yang sesuai dari mesin *Automatic LESSIMIC Analyzer,* dengan penjabaran sebagai berikut :

1. Hasil pengujian dengan aplikasi *Automatic LESSIMIC Analyzer* telah berhasil melakukan analisa leksikal, sintaksis, semantik juga kode antara dan *error handling* menggunakan bahasa pemrograman C#.
2. Hasil pengujian dengan menggunakan *Black Box testing* yang telah diuji coba untuk proses analisa leksikal, sintaksis, semantik juga kode antara dan *error handling* telah berhasil dilakukan sesuai dengan input yang diberikan.
3. Penelitian ini juga bertujuan untuk lebih memahami bagaimana proses kompilasi bekerja, sehingga kita selaku *programmer* mengetahui bagaimana computer memproses *code* yang kita *input* dan kita bisa membuat *source code* yang mangkus untuk di*compile* sehingga proses komputasi menjadi efisien.

## DAFTAR PUSTAKA

[1] Maukar, S. Lamria, and H. Widowati, "*Compiler* Sederhana Untuk Bahasa Pemrograman Sangat Sederhana," in *Seminar Nasional Sistem dan Informatika*, 2006, pp. 84–87.

[2] J. Suciadi, "Studi Analisis Metode-Metode Parsing dan Interpretasi Semantik Pada Natural Language Processing," *J. Inform.*, vol. 2, pp. 13–22, 2001.

[3] S. Slamet and H. Suhartanto, *Teknik Kompilasi*. Jakarta: Universitas Indonesia, 1992.

[4] S. N. Lailela, "Perancangan Perangkat Ajar Teknik Kompilasi," *J. CLICK*, vol. 1, pp. 74–87, 2017.

[5] V. V, S. M, and M. R S, "Paraphrase Identification in Short Texts Using Grammar Patterns," in *In 2013 3rd International Conference on Recent Trends in Information Technology, ICRTIT*, 2013, pp. 472–477.

[6] F. Utdirartatmo, *Teknik Kompilasi*. Yogyakarta: Graha Ilmu, 2005.

[7] M. Olivya, "Sistem Penilai Source Code Otomatis," pp. 44–55, 1998.

[8] Heriyanto, "Pendekatan *Compiler* Dengan Teori Bahasa dan Automata," *J. Ilm. STMIK AUB*, vol. 3, 2004.